Building Projects 13

Building a project is the final step of a long process. This chapter explains how to build five types of projects:

- Applications
- Desk accessories
- Device drivers
- Code resources
- Libraries

Contents

Putting a Project Together 255

Building Applications. 256
Setting the application file type and creator 257
Using a larger jump table with Far CODE 257
Using more global data with Far DATA 258
How Far CODE and Far DATA work 258
Mixing Near CODE and DATA with Far CODE and DATA . 258
Using a separate STRS component 259
Setting the partition size and SIZE resource flags 259

Building Desk Accessories and Device Drivers 262
Setting the project type. 262
How drivers work 264
How to write main( ) for a driver 265
Getting the event record pointer from paramBlock . . . 265
Global data in drivers 266
Using driver globals in callback and trap intercept routines 267
Using libraries in drivers 268
Setting the fields of a driver's header 268
Opening an open driver 271
How to return from a driver 271
The jIODone problem 272
Multi-Segment drivers 273

Building Code Resources 274
Setting the project type. 275
How to write main() for a code resource. 276

Global data in code resources 277
Using libraries in code resources 278
Locking code resources 279
Code resource headers 281
Merging code resources into files 283
Multi-segment code resources 283

Building Libraries 284
Projects as libraries 285
Binary libraries 285

Putting a Project Together

When you choose a **Build** command from the **Project** menu, the THINK Project Manager puts all the pieces of your project together as compactly as possible. The THINK Project Manager uses a technique called smart linking to extract only the CODE elements it needs to build your application, desk accessory, device driver, or code resource.

The THINK Project Manager searches the project for files or libraries whose code is never referenced and ignores them during the link. If you use a project as a library, the THINK Project Manager takes only the code it needs from the project. If you use a library that you built with the **Build Library** command, all of its code is included in the final file.

To learn how to build a library, see "Building Libraries" later in this chapter.

Smart linking yields smaller code, but it takes several seconds longer to produce the final file. You can choose not to use smart linking; your files are larger, but they come out faster.

Note

For desk accessories, code resources, and applications, the THINK Project Manager checks the components' size limits (described above) *after* smart linking.

To turn smart linking off, choose your **Build** command from the **Project** menu as you normally do, then click on the Smart Link check box to clear the option. (It's on by default.) See Figure 13-1.

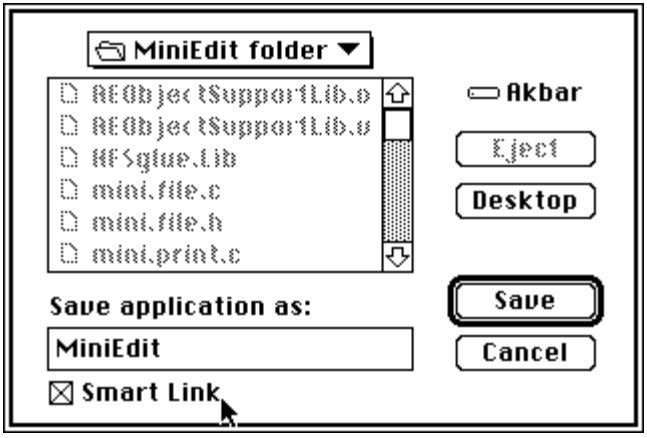

Figure 13-1 The Smart Link check box in a Build dialog

After the THINK Project Manager finishes linking the object code, it produces the application, desk accessory, device driver, or code resource file. Finally, it copies any resources in the project resource file into the final application file.

Building Applications

The THINK Project Manager is set up to build applications by default. The **Set Project Type** dialog gives you an opportunity to set

some application attributes. Choose **Set Project Type** from the **Project** menu, and you see the dialog in Figure 13-2:

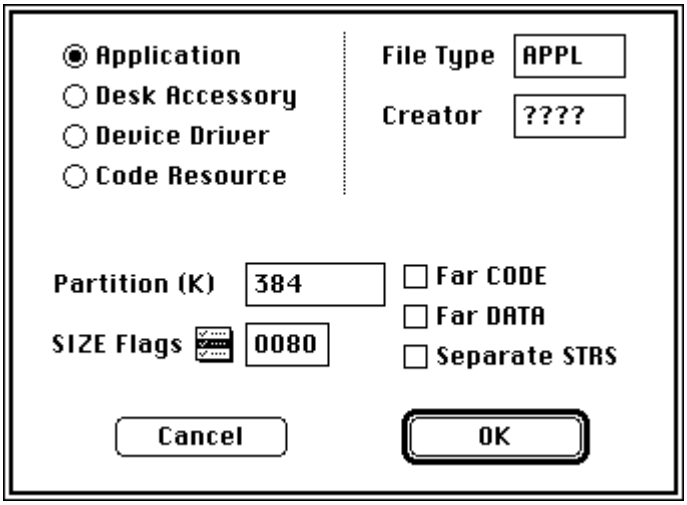

Figure 13-2 The Set Project Type dialog for applications

Setting the application file type and creator

To learn about File Types and Creators (also called signatures), see Inside Macintosh VI*, Chapter 9, "The Finder Interface."*

When you're building an application, the default file type is APPL. (For applications to work with the Finder, applications must be of type APPL.) Set the Creator to whatever you want your application's signature to be. The Finder uses the file's creator to link icons with specific applications.

Using a larger jump table with Far CODE

If you're building a large application, especially one that uses the THINK Class Library, its jump table can grow large quickly. The jump table contains entries for the following:

- Two entries for each virtual method
- One entry for each routine called by a routine in another segment
- One entry for the address of each routine that you use

If you need a large jump table, check the Far CODE option. It allows a jump table as large as 256K, since it uses 32-bit absolute addresses instead of 16-bit relative addresses. Your application will be about 6% larger because of the longer addresses. If the Far CODE option is off, your jump table can be only 32K.

Note

If the Far CODE option is on, each of your project's files can contribute only 4095 jump table entries (or 32K).

Using more global data with Far DATA

If you use a lot of global data, check the Far DATA option. It allows you to have an unlimited amount of global data, since it uses 32-bit absolute addresses instead of 16-bit relative addresses. Your application will be about 8% larger because of the longer addresses. If the Far DATA option is off, your project can contain only 32K of global variables.

Note

If the Far DATA option is on, each of your project's files can contribute only 32K of global data. This limit means, for example, that you can't use an array that's over 32K, unless you allocate the memory for it dynamically.

How Far CODE and Far DATA work

Far CODE and Far DATA work similarly. You can have larger applications with more space for your jump table and global variables, because the THINK Project Manager handles addresses into these areas differently. When these options are off, addresses into these areas are given as offsets from A5. In the MC68000 family, register-relative offsets must be 16-bits long, so these areas can contain only 32K. When the option is on, addresses to these areas are absolute addresses and can be 32-bits long. The THINK Project Manager includes code in your application that adds the value of A5 to these absolute addresses before your code runs.

Mixing Near CODE and DATA with Far CODE and DATA

If the Far CODE or Far DATA option is on and your application contains libraries that were compiled with those options off, put those libraries together in their own segment (or segments). The THINK Project Manager places the jump table entries for those libraries in the first 32K of the jump table. If the libraries are grouped together, the THINK Project Manager won't waste space in the first 32K by adding entries that can be elsewhere. Also, if one of those libraries calls a function in one of your files, put that file in the same segment as the libraries.

Note

When you precompile the header file `oops.h`, you must use the the same Far CODE and Far DATA option settings as the file or project that will use the header. For more information, see "Precompiling Headers" in Chapter 11.

Using a separate STRS component

THINK C normally places string literals in the DATA component of a project. When you check Separate STRS option, THINK C uses a separate STRS component to store string literals.

Note

Symantec C++ cannot store string literals in a separate component, and it ignores the setting of this option. To get more space for global data, use the Far DATA option, described in "Using more global data with Far DATA" on the previous page, or the Place String Literals in Code option, described in the *Symantec C++ Compiler Guide*.

When this option is off (the default), THINK C uses a 2-byte offset from register `A5` to access the string literals. Not only is the code smaller, but there's no need to do run-time address fix-ups. When the Separate STRS option is on, THINK C uses a 4-byte absolute address to access the string literals.

The Separate STRS option is useful only if you need a library compiled with a previous version of THINK C. If your project contains a library that was compiled with the Separate STRS option on, you must turn on the Separate STRS option in your project, too. If you need more space for global data, use the Far DATA option, described in "Using more global data with Far DATA" on the previous page.

Setting the partition size and SIZE resource flags

The THINK Project Manager uses the values you set in the Partition and SIZE Flags fields to build the `SIZE` resource your application needs to run under System 7 or MultiFinder.

The partition size determines how much memory your application gets. The default partition size is 384K, which is more than enough for moderate size applications. You'll want to use a higher value for larger applications or a lower value for smaller applications when memory is limited.

The SIZE Flags field sets the flags that tell System 7 and MultiFinder how compatible your application is and which features it uses. To learn how to write applications for System 7 or MultiFinder, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

You can type in the value of the flag (in hexadecimal) in the SIZE flags field, or you can set the bits from the pop-up menu next to the field. The pop-up menu is in Figure 13-3.

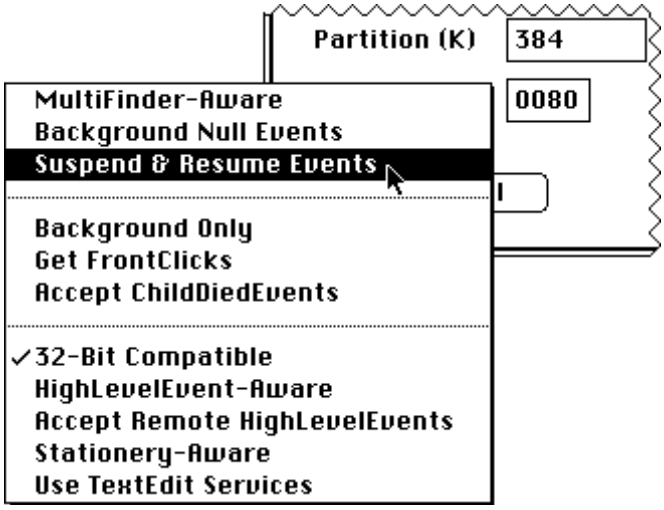

Figure 13-3 The SIZE Flags menu

The bits fall into three categories: basic MultiFinder support, extra MultiFinder features, and System 7 features. The first three bits describe how compatible your application is with System 7 and MultiFinder:

- If the MultiFinder-Aware bit is set, the Finder expects you to conform to the guidelines for shifting from the foreground to the background layers. Your application gets Suspend and Resume events as your application shifts from foreground to background; but it does not get acti-

vate/deactivate events. If MultiFinder Aware is checked, Suspend & Resume events should be checked as well.

- If the Background Null Events bit is set, your application gets regular null events when your application is in the background. Otherwise, your application gets only update events.

- If the Suspend & Resume Events bit is set, your application gets these events as it shifts from the foreground to the background layers in addition to the activate/deactivate events you normally receive.

The next three bits explain which of the extra features of System 7 and MultiFinder your application uses.

- If the Background Only bit is set, your application runs only in the background. Set this bit if your application has no user interface and cannot run in the foreground.

- If the Get FrontClicks bit is set, your application receives the mouse-down and mouse-up events that bring your application to the foreground.

- If the Accept ChildDiedEvents bit is set, your application is notified when an application it launched quit or crashed.

The last four bits tell the Finder which of these new System 7 features your application supports: 32-bit memory, AppleEvents, and stationery documents.

- If the 32-Bit Compatible bit is set, your application runs under the 32-bit version of Memory Manager. Don't set this flag unless you've tested your application on a 32-bit system, like a Macintosh IIci running System 7 in 32-bit mode or a Macintosh running A/UX.

- If the HighLevelEvent-Aware bit is set, your application receives all high-level events, including AppleEvents, when it calls `WaitNextEvent`. There is no way to mask out the types of high-level events your application can't handle. If this bit is not set, your application receives no high-level events.

- If the Accept Remote HighLevelEvents bit is set, your application receives high-level events, including AppleEvents, from your computer and other computers on your network. If this flag isn't set, your application receives high-level events from your computer only.

- If the Stationery-Aware bit is set, the Finder expects your application to handle stationery documents. If this bit isn't set, the Finder handles stationery documents for you by duplicating the document and asking the user for a name for the duplicate.

- Set the Use TextEdit Services bit if your application can use the inline text services that TextEdit provides for multi-byte script systems (such as those for Japan and China). See the chapter "TextEdit" in *Inside Macintosh VI* for more information about these services.

Building Desk Accessories and Device Drivers

Desk accessories and device drivers are structurally identical; they're both drivers. According to *Inside Macintosh*, drivers don't behave much like applications and have a different internal structure. In this section, the word driver by itself means either a device driver or a desk accessory.

This section won't teach you how to write a desk accessory or a device driver from scratch. To learn how to write them, read *Inside Macintosh I*, Chapter 14, "The Desk Manager," *Inside Macintosh II*, Chapter 6, "The Device Manager," and *Inside Macintosh V*, Chapter 23, "The Device Manager."

Note

For System 7, Apple recommends that you write small applications instead of desk accessories.

Setting the project type

Set the project type before you start working on a driver. If you set the project type after you've started compiling code, you must recompile your source files and reload your libraries.

Choose **Set Project Type** from the **Project** menu. When the project type dialog appears, click either the Desk Accessory button or the Device Driver button.

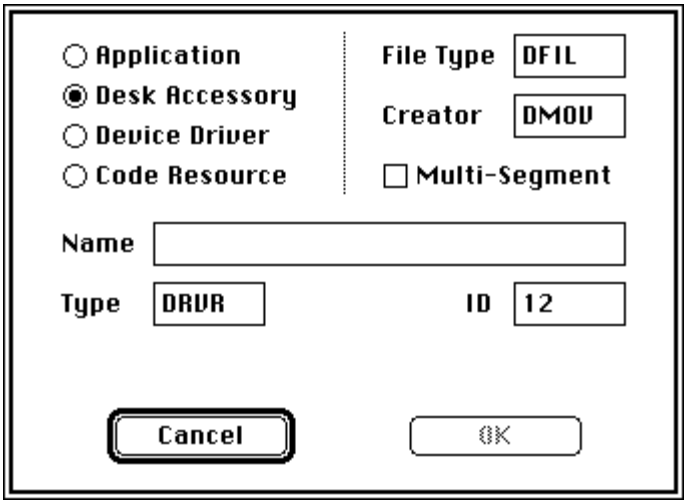

Figure 13-4 The Set Project Type dialog for desk accessories

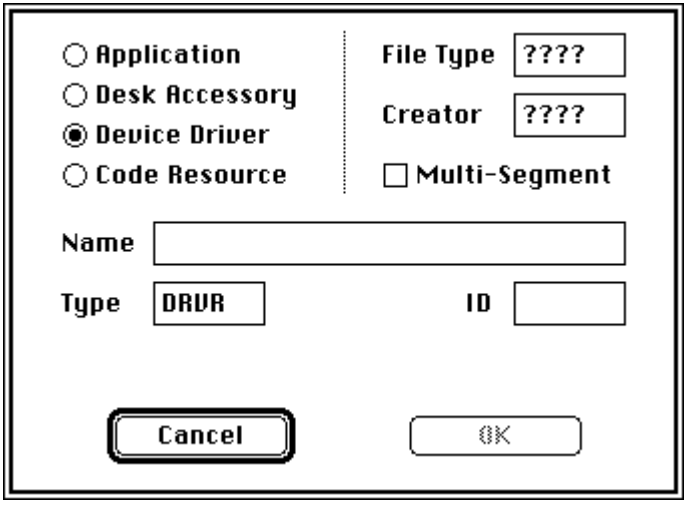

Figure 13-5 The Set Project Type dialog for device drivers

The dialog boxes are the same, except for the default settings of some fields. The **Desk Accessory** dialog presets the File Type and Creator so the resulting file is a suitcase file. The ID is set to 12. The

Finder renumbers it when you install it in your system. You shouldn't need to change the ID number. All that's left to do is to name the desk accessory and write the code. By convention, desk accessory names begin with a null. The THINK Project Manager provides the null for you automatically.

Note

In System 6, you need the Font/DA Mover to install desk accessories, since the Finder doesn't install fonts and desk accessories.

The **Device Driver** dialog leaves all the options empty for you to set. Device driver names begin with a period. If you don't provide one in the Name field, the THINK Project Manager automatically provides one for you.

The default settings in the project type dialogs aren't the only difference between desk accessories and device drivers. See "Setting the fields of a driver's header later in this chapter to learn about the internal differences between desk accessories and device drivers.

Both device drivers and desk accessories can have more than one segment, just like applications. Just click on the Multi-Segment check box. You can learn more about multi-segment drivers below, but first learn how a driver works.

Note

To use objects in a driver, you must turn on the Multi-Segment option, even if your driver has only one segment.

How drivers work

The Device Manager expects drivers to have five entry points and to be written in assembly language. When the Device Manager calls a driver written with the THINK Project Manager, a short assembly-language stub (or glue routine) translates the Device Manager's request into a call to the C function `main()`. The THINK Project Manager automatically places the driver glue at the beginning of your driver.

The THINK Project Manager does two things to make writing a driver easier. First, it sets up a data area so that your driver can have its own global variables. Second, it figures out the proper way to return control to the Device Manager automatically. These services are discussed later in this section.

How to write main( ) for a driver

The driver's `main()` function takes three arguments. The function returns a `short int` and is *not* declared `pascal`. A typical driver skeleton looks like this:

```
short main (CntrlParam *paramBlock,
   DCtlPtr devCtlEnt, short n)
{
   switch (n) {
      case 0 :    // Open
      case 1 :    // Prime
      case 2 :    // Control
      case 3 :    // Status
      case 4 :    // Close
   }
}
```

`MacHeaders` *and* `MacHeaders++` *include the files* `Files.h` *and* `Devices.h` *by default.*

`paramBlock`, defined in `Files.h`, is a pointer to an I/O parameter block. This is the value that is passed in address register `A0` to the assembly-language entry point of the driver.

`devCtlEnt`, defined in `Devices.h`, is a pointer to the driver's device control entry. This is the value that is passed in address register `A1` to the assembly-language entry point of the driver.

`n` is a selector that specifies which entry point actually received the call. Use the value of `n` to dispatch control to the appropriate routine.

Getting the event record pointer from paramBlock

According to *Inside Macintosh I*, Chapter 14, "The Desk Manager," the `csCode` field of the `paramBlock` passed to your driver specifies what kind of action your driver should take.

When `paramBlock->csCode == accEvent`, the `csParam` field of `paramBlock` contains a pointer to an `EventRecord`. Since `csParam` is defined as an array of `int`s, this is how you cast the

field to get a pointer to the event record, assuming that `eventPtr` is declared `EventRecord *`:

```
eventPtr =
   (*(EventRecord **)paramBlock->csParam)
```

Global data in drivers

You can declare global and static variables in drivers. The THINK Project Manager driver glue allocates the space for the globals in the heap before it calls `main()` to implement the `Open` entry. The glue releases the memory when the driver returns from a `Close` call. (There is a way to keep the global data area allocated after a `Close`; see "How to return from a driver later in this chapter.)

Macintosh applications use register `A5` to access their globals. Since drivers coexist with running applications, they can't use register `A5` to access their globals. Instead, drivers use register `A4`.

Note

In drivers, string literals are usually stored the same way as global variables. However, in C++ code when the Place String Literals in Code option is on, string literals are stored in the generated code and are always accessed off the `PC`. For more information, see *Symantec C++ Compiler Guide*.

The THINK Project Manager driver glue stores a handle to the dynamically allocated data area in the `dCtlStorage` field of the driver's device control entry. This handle is dereferenced into address register `A4` and locked before each call to `main()`. Your `Open` routine must check whether the data area was allocated successfully. If it was not, the `dCtlStorage` field is 0, and your driver should display some error message (without using any globals!) and close itself.

The data area remains locked between calls to your driver. If you like, you can unlock it yourself before returning. If you unlock the data area, though, make sure that you don't rely on the address of any data item staying the same between calls. Also, make sure that the data area doesn't contain any objects, such as windows, that the Toolbox assumes do not move.

Using driver globals in callback and trap intercept routines

The THINK Project Manager driver glue sets up register A4 for you whenever it's called from `main()`. You have to save A4 where your routines can find it, if your driver defines callback routines, trap intercept routines, or other functions that might be called when the value of A4 is in doubt.

The header file `SetUpA4.h` defines a set of macros that take care of saving, setting, and restoring A4 for you.

Suppose your driver calls `ModalDialog()` with a `filterProc`. Since you're not sure if the value of A4 is correct when `ModalDialog()` calls your `filterProc`, you need to save it. Your `filterProc` needs to set A4 to the saved value and then restore it before it exits. Your call to `ModalDialog()` would look like this:

```
engage_in_dialog()
{
   extern pascal Boolean myFilter();
   short int item;
   …
   RememberA4();
   ModalDialog(myFilter, &item);
   …
}
```

Your `filterProc` would look like this:

```
pascal Boolean myFilter(DialogPtr dp,
                        EventRecord eventp,
                        short *itemp)
{
   Boolean result;

   SetUpA4();
   …
   RestoreA4();
   return(result);
}
```

The calls to `RememberA4()` and `SetUpA4()` must appear in the same source file.

Note

If your routine uses an aggregate initializer for a local variable, that variable must be declared `static` or it will have the wrong value.

Use the same technique for trap intercept routines that need access to a driver's globals. Of course, if your callback or trap intercept routine doesn't use driver globals, you don't need to set up and restore `A4`.

Using libraries in drivers

You can use libraries in drivers as long as the libraries don't reference global variables accessed through register `A5`.

The `MacTraps` and `MacTraps2` libraries don't reference any globals, so you can use them in your drivers. `MacTraps` does define the QuickDraw globals, though. If you access these globals from a driver, you'll find that they aren't the real QuickDraw globals but simply the driver's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a driver by observing that `0(A5)` holds the address of the last of the QuickDraw globals, `thePort`. The remaining QuickDraw globals are at descending addresses from `thePort`. Refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of `A5` is stored in the low-memory global `CurrentA5`. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 20, "Writing Inline Assembly," for details.

Other libraries supplied with THINK Project Manager compilers reference their globals from register `A5`, so you must modify them. To make a library use register `A4` for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library. See "Building Libraries later in this chapter to learn how to build libraries.

Setting the fields of a driver's header

A driver begins with a header containing several flags and other data items (some of which apply only to desk accessories). When the driver is opened, the Device Manager copies these fields to the device control entry before the `Open` entry point is called. After that,

the device header is not used. The fields in the driver are used only to initialize the fields in the device control entry.

The THINK Project Manager presets these fields to reasonable default values. If your device driver or desk accessory requires different settings for these fields, modify them on the fly in the device control entry.

These are the default settings for desk accessories:

Field	Value	
dCtlFlags	dReadEnable	0
	dWritEnable	0
	dCtlEnable	1
	dStatEnable	0
	dNeedGoodbye	0
	dNeedTime	0
	dNeedLock	0
dCtlDelay	N/A because dNeedTime = 0	
dCtlEMask	0x016A (mouseDown, keyDown, autoKey, update, activate)	
dCtlMenu	0	

Table 13-1 Default header for desk accessories

These are the default settings for device drivers:

Field	Value	
dCtlFlags	dReadEnable	1
	dWritEnable	1
	dCtlEnable	1
	dStatEnable	1
	dNeedGoodbye	0
	dNeedTime	0
	dNeedLock	1
dCtlDelay	N/A because dNeedTime = 0	
dCtlEMask	N/A (desk accessories only)	
dCtlMenu	N/A (desk accessories only)	

Table 13-2 Default header for drivers

Suppose you want a desk accessory to remain locked between calls and to be called once per second (every 60 ticks). Just include this

code in the driver's `Open` routine. (`devCtlEnt` is the second argument to `main()`, the pointer to the device control entry.)

```
devCtlEnt->dCtlFlags|= dNeedLock|dNeedTime;
devCtlEnt->dCtlDelay = 60;
```

Opening an open driver

The `Open` entry point of a driver (the third argument of `main()` is 0) may be called even if the driver is already open. This happens, for example, when the user selects the name of a desk accessory that's already on the screen. The driver should check to see if it is already open to avoid repeating its initialization sequence.

You can set the fields of the device control entry directly as shown above, but the Device Manager copies the `dCtlFlags`, `dCtlMenu`, `dCtlDelay`, and `dCtlEMask` fields of the driver's header to the corresponding fields of the device control entry *every time* the `Open` routine is called, even if the driver is already open. So you need to set these fields to their proper values each time. Your `Open` routine might look something like this:

```
doOpen()
{
   devCtlEnt->dCtlFlags|= dNeedLock;
                       // or whatever
   if (already_open) // already_open is a
      return;        //   driver global
   already_open = 1; // one-time
}                     //   initialization.
```

How to return from a driver

If your `Open` routine was successful, return `0`. If the `Open` routine fails, return a negative result and the driver isn't opened.

Return `0` from `Close` if it was successful. If your `Close` routine returns `closeErr` (`-24`) the driver won't be closed. If you return a `1`, the THINK Project Manager driver glue preserves the `dCtlStorage` field of the device control entry. This way you can keep your driver globals around until your driver is reopened. (The driver glue makes it seem as though your `Close` routine returned `0`, meaning the `Close` was successful.).

Warning

Returning negative values from `Open` and `Close` to prevent opening or closing works only on 128K and later ROMs.

Return `1` from asynchronous calls to `Prime`, `Control`, and `Status` routines if the request could not be completed right away. This result code is stored in the `ioResult` field of the I/O parameter block, but `0` (no error) is returned to the Device Manager.

The jIODone problem

The THINK Project Manager always returns from a driver correctly. In other development systems, it's not so easy. This section deals with development systems in which returning from a driver can be a problem.

One of the trickiest aspects of returning from a driver is deciding whether to return directly to the Device Manager (via an `RTS` instruction) or whether to jump to `jIODone`. This is a complex issue, and many existing desk accessories do it wrong (though, fortuitously, they manage to work anyway).

Associated with each driver is an I/O queue, which is a list of I/O parameter blocks waiting for service from the driver. Calls made to a driver fall into one of two categories: "queued," meaning that the I/O parameter block passed as an argument to the call is in the driver's queue; and "immediate," meaning that it is not. In the immediate case, the queue may even be (and in fact usually is) empty.

All `Open` and `Close` calls are immediate. All `Control` calls made to desk accessories are immediate, except for the "good-bye kiss" (when `csCode` is `-1`) issued to desk accessories that have requested to be notified when the current application exits out from under them. Other calls may be queued or immediate.

The rules for returning from a driver are that the driver should return directly to the Device Manager from all immediate calls. It should also return directly to the Device Manager from queued calls requesting asynchronous I/O that could not be completed right away. Finally, it should jump to `jIODone` from queued calls if the driver completed the request (or if there was an error).

It is incorrect to violate these rules; in particular, it is incorrect to jump to `jIODone` to return from an immediate call. `jIODone` attempts to examine the driver's I/O queue, and since the queue is usually empty it ends up examining low-memory locations beginning at `0x0000`. Apparently, these locations somehow look enough like an I/O parameter block to satisfy the Device Manager, but this is clearly an unsafe situation.

When returning from `Prime`, `Control`, and `Status` calls, `jIODone` unlocks the driver's code and its device control entry so they won't form islands in the heap between calls to the driver (unless, of course, the driver has requested that they remain locked). So the author of a desk accessory, has to make a difficult decision — to return directly to the Device Manager, leaving the driver's code and its device control entry locked and potentially interfering with the host application; or to violate the rules and jump to `jIODone`. Most authors of desk accessories seem to take the latter route.

The THINK Project Manager avoids this dilemma. When a driver written with the THINK Project Manager returns from `main()`, the decision whether to call `jIODone` is made automatically (and correctly). For `Prime`, `Control`, and `Status` calls, if the decision is made to return directly to the Device Manager, and the driver has not requested that its code and device control entry remain locked, they are unlocked.

Multi-Segment drivers

If the Multi-Segment option is on, drivers can contain multiple segments. As with applications, segments are loaded automatically as they are called. In addition, all loaded segments are unloaded automatically upon return to the Device Manager after each call, *unless* the `dNeedLock` bit is set in the driver's device control entry.

You can unload driver segments manually with this function:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications.

Warning

Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

To learn how to break up a project into different segments, see "Segmenting Projects" in Chapter 8.

When the THINK Project Manager creates a multi-segment driver, it puts the code segment that contains the `main()` function in a `DRVR` resource. This is called the owning resource. You set its ID in the **Set Project Type** dialog, which must be between 0 and 63 (inclusive). The THINK Project Manager puts the other code segments in `DCOD` resources and numbers them from 0 to 31. These are called owned resources.

Building Code Resources

You can use the THINK Project Manager to write pure code resources. Code resources don't have the complex structure of drivers; they simply contain code to be called at the entry point, `main()`.

For more information on calling conventions, see "Calling C Routines" and "Calling Pascal Routines" in Chapter 19.

You might want to write code resources for several reasons. You might want to write a window definition function that you can use in several other programs, or you might want to write a system extension that runs at start-up. You may define your own code resource types to make a function you've written with the THINK Project Manager available to a program written in another language. The "client" program simply loads the resource and calls it at its beginning. It's up to you whether you use C or Pascal calling conventions.

This section tells you how to build code resources with the THINK Project Manager. The specific formats and calling sequences for code

resources are given in the various volumes and chapters of *Inside Macintosh*. This list helps you get started.

To build this…	Read *Inside Macintosh*…
ADBS	Vol. V, Chap. 20, “The Apple Desktop Bus”
CDEF	*Macintosh Toolbox Essentials*, “Control Manager”
cdev	*Macintosh Toolbox Essentials*, “Control Manager”
FKEY	Technical Note 3 (also see below)
INIT	Vol. IV, Chap. 29, “The System Resource File” Vol. V, Chap. 19, “The Start Manager”
LDEF	*More Macintosh Toolbox*, “List Manager”
MBDF	*Macintosh Toolbox Essentials*, “Menu Manager”
MDEF	*Macintosh Toolbox Essentials*, “Menu Manager”
WDEF	*Macintosh Toolbox Essentials*, “Window Manager”
XCMD	HyperCard Script Language Guide
XFCN	HyperCard Script Language Guide

Table 13-3 Code resource information

For more information on the Pascal Object model, see the Symantec C++ Compiler Guide.

Note

A code resource created with Symantec C++ cannot contain classes created with the standard C++ object model that use virtual functions. However, it can contain classes created with the Pascal Object model that use virtual functions.

Setting the project type

Set the project type before you start working on a code resource. If you set the project type after you’ve started compiling code, you must recompile your source files and reload your libraries.

Choose **Set Project Type** from the **Project** menu. When the project type dialog appears, click on the Code Resource check box.

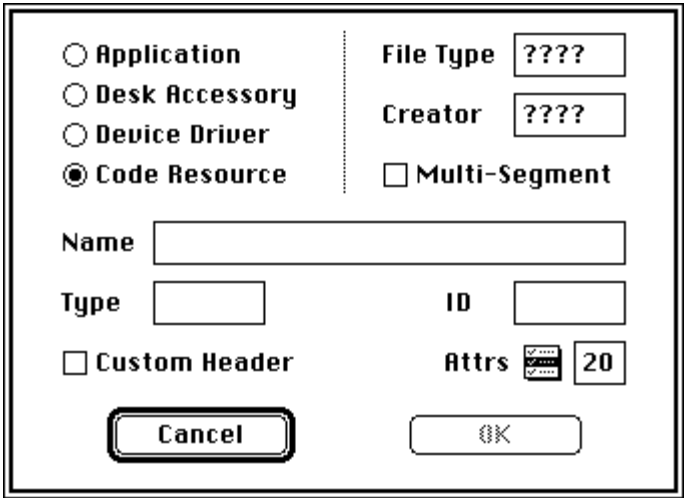

Figure 13-6 The Set Project Type dialog for code resources

Fill in the Type and ID of the code resource you're building. If you like, you can give your code resource a name. Use the Attrs (attributes) pop-up menu to set the resource attributes for your code resource. If you prefer, you can enter a hex value in the Attrs field. To learn about resource attributes, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

Code resources can have more than one segment. If your code resource has multiple segments, select the Multi-Segment option. Read the section "Multi-segment code resources" later in this chapter for more information.

Warning

To use objects in a code resource, you must turn on the Multi-Segment option, even if your code resource has only one segment.

How to write main() for a code resource

The way you write your `main()` routine depends on the kind of resource you're writing. The main point to remember is that if your code resource is going to be called from a Pascal program or from a callback routine, it must be declared `pascal`.

An FKEY resource, for example, is called by the Event Manager. It doesn't have any arguments. You would define `main()` like this:

```
main()
{
    …
}
```

A WDEF resource is a custom window definition. The Window Manager calls `main()` with several arguments and expects the window definition function to return a `long`. This is how you would write `main()` for a WDEF:

```
pascal long main (short varCode,
                  WindowPtr theWindow,
                  short message,
                  long param)
{
    …
}
```

Global data in code resources

Code resources, like applications and drivers, can have global and static variables.When you build a single-segment code resource, the DATA component is appended to the CODE component, so CODE and DATA together must be less than 32K. When you build a multi-segment code resource, the DATA and JUMP components from all the segments are appended to the CODE component of the segment that contains `main()`, so all the DATA and JUMP information and the CODE component for the main segment must be less than 32K.

Code resource globals are addressed as offsets from A4. Unlike drivers, however, A4 isn't set up automatically for you when your `main()` routine is called. You have to do this yourself.

Note

If your `main()` routine uses an aggregate initializer for a local variable, that variable must be declared `static` or it will have the wrong value.

Warning

In multi-segment code resources, THINK C stores string literals the same way as globals. If you're using literals and constants but not globals, you still need to set up A4. In single-segment code resources, string literals are stored in code and you don't need to set up A4 to use them. Symantec C++ handles string literals in code resources as THINK C does, unless you turn on the Put String Literals in Code option. Then, Symantec C++ always stores string literals in code, and you don't need to set up A4 to use them.

When `main()` is called, A0 points to your code resource. This is the same value that your code resource expects A4 to have to find your globals.

The header file `SetUpA4.h` contains a set of macros that help you set up the A4 register. Immediately after you enter `main()`, call `RememberA0()`. This macro saves the value of A0 where another macro, `SetUpA4()`, can find it. You must call `RestoreA4()` before you return from `main()`. For example:

```
main()
{
    RememberA0(); // To access globals
    SetUpA4();
    /*
     * Your resource does its work here.
     */
    RestoreA4();
}
```

Note

This technique works only when you use the default code resource header. If you use a custom header, you must set up A4 differently. See "Code resource headers" later in this chapter for more information.

Using libraries in code resources

You can use libraries in code resources as long as the libraries don't reference global variables accessed through register A5.

The MacTraps and MacTraps2 libraries do not access any globals, so you can use them in your code resources. MacTraps does define the QuickDraw globals, though. If you access these globals from a code resource, you'll find that they aren't the real QuickDraw globals but simply the resource's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a code resource by observing that 0(A5) holds the address of the last of the QuickDraw globals, thePort. The remaining QuickDraw globals are at descending addresses from thePort. Refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of A5 is stored in the low-memory global CurrentA5. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 20, "Writing Inline Assembly," for details.

For more information on how to create libraries, see "Building Libraries" later in this chapter.

Most libraries in the Standard Libraries folder reference their globals from register A5, so you must recompile them. (Libraries whose names end in -A4, such as ANSI-A4, reference their globals from A4.) To make a library that uses register A4 for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library.

Locking code resources

The Macintosh Toolbox takes care of locking and unlocking the standard code resources like WDEFs. When you write your own code resources, you can either let the caller take responsibility for locking and unlocking them, or you can have the code resource do it.

When `main()` is entered, register `A0` contains a pointer to your code resource. If you need to lock it, you would write `main()` like this:

```
#include <SetUpA4.h>
#include <Types.h>
#include <Memory.h>

#pragma parameter __D0 RecoverResPtr()
Handle RecoverResPtr() = 0xA128;
                    // Return the contents
                    // of A0.

void main()
{
   Handle h;

   RememberA0();    // Set up A4 so you
   SetUpA4();       // can access resource
                    // globals.

   h = RecoverResPtr(); // Get pointer to
   HLock(h);            // resource and
                        // lock it.

   /*
    * The code resource does its work here.
    */

   HUnlock(h);
   RestoreA4();
}
```

Warning

This technique works only when you use the standard code resource header. If you use a custom header, you must get the address of your code resource another way. See “Code resource headers” below.

If your code resource can be called reentrantly, it should not be unconditionally unlocked each time it returns. Instead, it should be restored to the same locked state it had on entry.

Code resource headers

If the Custom Header option is off, the THINK Project Manager creates a code resource with the standard header in Table 13-4.

Offset	Contents
0	`BRA.S .+0x10` (branch to header code)
2	`0x0000` (unused)
4	`'TYPE'` (resource type)
8	`0x000A` (resource ID)
10(0xA)	`0x0000` (unused)
12(0xC)	`0x0000` (unused)
14(0xE)	`0x0000` (unused)

Table 13-4 Standard code resource header

The standard header code puts the address of your code resource in register `A0` and then branches to your `main()` routine, but the file containing `main()` is *not* guaranteed to be the first file in the code resource. You can do anything you like with the unused words.

Note

Older versions of THINK C put the address of your code resource in the low memory global `ToolScratch`. The standard code resource header does not do this. Use inline assembly if you need to reference `A0` directly.

If the Custom Header option is checked, the THINK Project Manager does not generate the standard header. Instead, your code resource starts with the first function in the file where `main()` is defined. The routine `main()` doesn't have to be the first function in the file, so your resource can begin any way you like. (In fact, `main()` may never be called.) This option is useful for certain types of code resources that must begin with a table of some kind, rather than with code.

When you use a custom resource header, `A0` does not contain the address of your code resource. This means that you can't use the `RememberA0()` macro to set up register `A4` to use your code resource globals until you set up `A0` to point to your code resource.

Warning

Do not include `SetUpA4.h` at the top of your file, because `SetUpA4.h` generates code and its internal function is your header.

The following code shows one way to set up your code resource globals when you use a custom resource header.

```
/*
 * Put #includes, globals, and
 * declarations here
 */

extern main();          // Declare main() so
                        //  you can JMP to it

header()                // header() is first
{                       //  function in file
   asm {
   DC.L  0              // Header information
   // . . .
   // End of header

   LEA   header, a0     // Put address of code
                        //  resource in A0.
   JMP   main           // Jump to main().
   }
}

/*
 * SetUpA4.h generates code, so don't put
 * it at the top of the file with the other
 * headers
 */
#include <SetUpA4.h>

main()
{
   RememberA0();
   SetUpA4();

   /*
    * The code resource does its work here.
    */

   RestoreA4();
}
```

If you want to use a custom header in a Symantec C++ code resource, you'll probably want to write the custom header in a THINK C file so you can use the THINK C inline assembler. To learn how to link C and C++ files together, see the *Compiler Guide*.

Merging code resources into files

Unlike other project types, you can build code resources into already existing files. In the **Build Code Resource** dialog, check the Merge check box, and type the name of the file you want your code resource placed into.

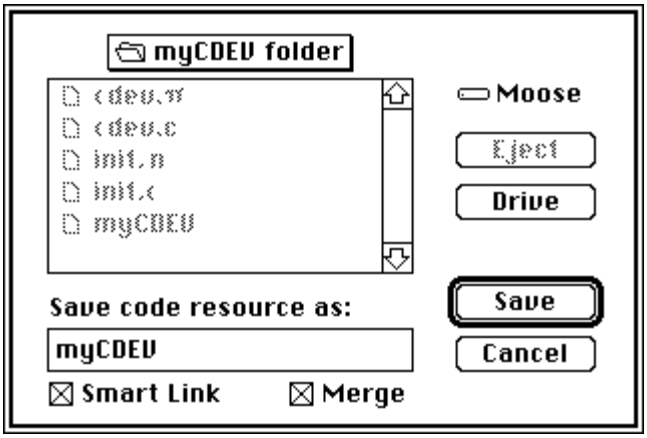

Figure 13-7 The Build Code Resource dialog

You might want to do this when you're building a library of `XCMD`s or building a combination `cdev` and `INIT`. However, when you check the Merge check box, the project's resource file is not be copied into the file, so you should make sure that any resources needed by your code resource are already present in the file.

Warning

Don't merge code resources into your System file. You could corrupt it.

Multi-segment code resources

To learn how to segment a project, see "Segmenting Projects" in Chapter 8.

If the Multi-Segment option is checked in the **Set Project Type** dialog, code resources can contain multiple segments. As with the other project types, segments are loaded automatically as they are called. However, unlike the other project types, you must unload the

segments yourself when the code resource exits. You can unload individual segments with this function:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications. To unload all segments at once, call `UnloadA4Seg()` with a null pointer; that is, `UnloadA4Seg(0L)`.

Warning

Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

In a multi-segment code resource, the code segment that contains the `main()` function is called the owning resource. The other segments are called owned resources. When the THINK Project Manager builds your resource, it uses the Type and ID number from the **Set Project Type** dialog for the owning resource. The ID of the owning resource must be 63 or lower. You can change this ID after you build the resource. The THINK Project Manager puts the owned code segments in Method declarations and invocations `'CCOD'` resources and numbers them from 0 to 31. You cannot change these numbers after you build the resource.

In code resources, the owning segment is special. The DATA and JUMP components for all the segments are appended to the CODE component of the owning segment. So all the DATA and JUMP information and the CODE component of the main segment must be under 32K.

Building Libraries

The THINK Project Manager gives you two ways to create libraries. You can use any THINK Project Manager project as a library, or you can create a separate library document called a binary library. When you use a project as a library, the smart linker uses only the code it needs. Binary libraries, on the other hand, are smaller, and take up less space on disk.

Most of the libraries in your THINK Project Manager package are actually THINK Project Manager projects. The binary library format is included for compatibility with older versions of THINK C.

Projects as libraries

You can use any THINK Project Manager project document as a library. Libraries can contain functions, globals, or other libraries. When you use a project as a library, the THINK Project Manager uses smart linking to build your project. The linker links only the CODE components that contain functions that your program uses.

See Chapter 8, "Organizing Projects," to learn about the different components of a THINK Project Manager project.

If your library contains global variables, and you want to use it in code resources, desk accessories, or drivers, you need to make sure that you compile everything in the library with A4 addressing. To make THINK Project Manager compile your globals with A4-relative addressing, choose **Set Project Type** from the **Project** menu and click on the Code Resource radio button.

Note

When you choose Code Resource in this case, it doesn't mean that you're building a code resource or that the library will only work with code resources. It's just a way of letting the compiler know that you want A4-relative globals.

If your library doesn't contain any global variables, you can use it with either applications or code resources, drivers, or desk accessories. If your library does contain global variables, you'll need to create two versions: one for applications and one for everything else.

The ANSI libraries that come with your THINK Project Manager package are good examples of project libraries. If you want to explore them, make a copy of them first so you don't damage the original ones accidentally. Then use the **Open Project** command in the **Project** menu to open them.

Binary libraries

Use the **Build Library** command in the **Project** menu to create a binary library. When you use this command, you'll see a standard file dialog asking you to name the library. By convention, libraries end in `.lib`. When you use a library in your project, the linker adds all of the object code in the library to your project.

Binary libraries are seldom used. The facility to use and create them is provided for compatibility with older versions of THINK C and some third-party products.